Recap...

Let’s go over some topics we covered previously...
Parameters and return types

// adapted from Java Foundations p189
public char calc (int num1, int num2, String message) {
    int sum = num1 + num2;
    char result = message.charAt (sum);
    return result;
}

// passing an array as a parameter
public void printNames(String[] names) {
    for(int i=0; i < names.length; i++){
        System.out.println(names[i]);
    }
}

// returning an array
public int[] powersOf2(int iterations) {
    int[] output = new int[iterations];
    for(int i=0; i < iterations; i++){
        output[i] = Math.pow(2,i);
    }
    return output;
}
Arrays of Objects

collection[0] = new CD("Storm Front", "Billy Joel", 14.95, 10);
collection[1] = new CD("Come On Over", "Shania Twain", 14.95, 16);
collection[2] = new CD("Soundtrack", "Les Miserables", 17.95, 33);
Arrays of Objects

collection[0] = new CD("Storm Front", "Billy Joel", 14.95, 10);
collection[1] = new CD("Come On Over", "Shania Twain", 14.95, 16);
collection[2] = new CD("Soundtrack", "Les Miserables", 17.95, 33);

• What is the type of collection?
  CD[]

• How was the collection initialized?
  collection = new CD[100];

• What would collection[10] return?
  null
Arrays of Objects

• All values in a particular array have the same type, or are at least of compatible types.

Can Shape[] contain circles, triangles, and rectangles?

Yes! We will see this later with polymorphism.
Abstract classes

When you want to tell your children what to do, but you do not want to do it for them!
Need for Abstract Classes

Abstract classes and methods are shown in italics font.
Abstract Classes

• An *abstract class* is a **placeholder** in a class hierarchy that represents a **generic** concept

• An abstract class **cannot** be instantiated

• To declare a class as abstract:

```java
public abstract class Shape
{
    // contents
}
```

• Abstract classes are an important element of software design: they allow us to establish **common** elements in a hierarchy that are too **generic** to instantiate
Abstract Classes: Rules

• An abstract class often contains abstract methods with **no definitions**
  • The **abstract** modifier **must** be applied to each abstract method

• An abstract class typically contains non-abstract methods with full definitions

• A class declared as abstract **does not have to** contain abstract methods – simply declaring it as abstract makes the class abstract

• The child of an abstract class **must override** the abstract methods of the parent, or it, too, will be considered abstract

• An abstract method **cannot** be defined as final or static
Lab 3 - The shapes hierarchy

Draw the UML

- Circle
- Triangle
- Rectangle
- Square
Create an array of 3 circles

Circle[] circles = new Circle[3];
circles[0] = new Circle(3);
circles[1] = new Circle(5.8);
circles[2] = new Circle(0);

What would `System.out.println(circles[1]);` produce?

********** Testing Circle class **********
circle - area: 105.683 perimeter: 36.442 (radius: 5.8)
THE TRIANGLE CLASS

https://cs.wellesley.edu/~cs230/labs/?lab=3&part=1

• Create a new class Triangle:
  • A triangle is defined by its three sides, let's call them a, b, and c
  • Its perimeter is simply the sum of its three sides
  • Its area can be computed using Heron's formula:
    • $s = (a+b+c)/2$
    • $area = \sqrt{s(s-a)(s-b)(s-c)}$

• Task 1A: where should the Triangle class be placed in the Shapes hierarchy? Add a main().

• Task 1B: Determine what instance variables Triangle should contain. Create a constructor.

• Task 1C: Overwrite the missing methods.
/**
 * FILE NAME: Isosceles.java
 * WHO: Copyright Stella K. 2016
 * WHAT: Class with Isosceles Triangle properties.
 * Inherits from Triangle (which inherits from Shape) class
 */

public class Isosceles extends Triangle {

/**
 * Constructor
 * @param base The length of the base of the triangle
 * @param scelos The length of the each of the two equal legs of the triangle
 */

public Isosceles(double base, double scelos) {
    super (base, scelos, scelos); // call the constructor in the super class (Triangle)
    // which in turn calls the Shape constructor
    this.name = "Isosceles Triangle"; // now fix the name, since it is still "Triangle"
    // We can do this because the instance variable name is protected in the Shape class
}

public static void main(String[] args) { ... }
}
Polymorphism

Inheritance provides Power to OOP

Polymorphism provides flexibility through inheritance
Polymorphism via Inheritance

Class `Rectangle` has a method called `area()`, and the child class `Square` overrides it.

Now consider the following invocation:

```java
myShape.area();
```

Which `area()` is invoked?

If `myShape` refers to a `Rectangle` object, it invokes the `Rectangle` version of `area()`.

If `myShape` refers to a `Square` object, it invokes the `Square` version of `area()`!
Static and Dynamic Binding

• Consider the following method invocation:

  ```java
  myShape.area();
  ```

• At some point, this invocation is bound
definition of the method that it invokes

  • If this binding occurred statically at compile time,
    then that line of code would call the same method every time

• Java defers method binding until run time:
  this is called dynamic binding or late binding

• Dynamic binding provides flexibility in program design
Polymorphism: "having many forms"

• A *polymorphic reference* is a variable that can refer to different types of objects at different points in time

• Suppose we create the following reference variable

  ```java
  Rectangle myShape;
  ```

• Java allows this reference to point to a Rectangle object, or to any object of *any compatible type*!

• This *compatibility* can be established using *inheritance* or using *interfaces*
CS230 feedback